Personal Autonomic Computing Reflex Reactions and Self-Healing

Roy Sterritt, *Member, IEEE,* and David F. Bantz, *Member, IEEE*

***Abstract*—The overall goal of this research is to improve the self-awareness and environment-awareness aspect of personal autonomic computing (PAC) to facilitate self-managing capabilities such as self-healing. Personal computing offers unique challenges for self-management due to its multiequipment, multisituation, and multiuser nature. The aim is to develop a support architecture for multiplatform working, based on autonomic computing concepts and techniques. Of particular interest is collaboration among personal systems to take a shared responsibility for self-awareness and environment awareness. Concepts mirroring human mechanisms, such as reflex reactions and the use of *vital signs* to assess operational health, are used in designing and implementing the PAC architecture. As proof of concept, this was implemented as a self-healing tool utilizing a pulse monitor and a vital signs health monitor within the autonomic manager. This type of functionality opens new opportunities to provide self-configuring, self-optimizing, and self-protecting, as well as self-healing autonomic capabilities to personal computing.**

***Index Terms*—Autonomic computing (AC), environment aware, personal autonomic computing (PAC), personal computing, self-aware, self-healing, self-managing systems.**

I. Introduction

PERSONAL autonomic computing (PAC) is autonomic computing (AC) [1] in a personal computing environment [2]. Personal computing has evolved substantially into a consumer product. Its scope now extends from end user computing in the office to home PCs, wireless laptops, palmtops and is evolving into applications of personal embedded computing, for instance with next-generation mobile/cell phones and iPods. In the near future, these will be leaf nodes in the self-managing ubiquitous and pervasive computing environments incorporating next-generation internet.

Manuscript received October 15, 2004; revised April 25, 2005. This work was supported in part by the Centre for Software Process Technologies (CSPT), funded by Invest NI through the Centres of Excellence Programme, under the EU Peace II initiative. This work was presented in part at the Proceedings of the IEEE Workshop on the Engineering of Autonomic Systems (EASe 2004), 11th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2004), Brno, Czech Republic, May 24–27, 2004; DEXA 2004 Workshops, 2nd International Workshop on Self-Adaptive and Autonomic Computing Systems (SAACS 04), Zaragoza, Spain, Aug. 30–Sep. 3, 2004;Workshop on the Engineering of Autonomic Systems (EASe 2005), 12th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2005), Greenbelt, MD, Apr. 3–8, 2005. Part of this work was carried out while D. F. Bantz was at IBM TJ Watson Center.

R. Sterritt is with the School of Computing and Mathematics, Faculty of Engineering, University of Ulster, Jordanstown Campus, Newtownabbey, U.K. (e-mail: r.sterritt@ulster.ac.uk).

D. F. Bantz is with the Computer Science Department, School of Applied Science, Engineering and Technology, University of Southern Maine, Portland, ME 04104 USA (e-mail: bantz@cs.usm.maine.edu).

Digital Object Identifier 10.1109/TSMCC.2006.871592

Personal computing is an area that can benefit substantially from autonomic principles. Examples of current difficult experiences that can be overcome by such an approach include [2]: 1) trouble connecting to a wired or a wireless network at a conference, hotel, or other work location; 2) switching between home and work; 3) losing a working connection (and shouting across the office to see if anyone else has had the same problem!); 4) going into the IP settings area in Windows and being unsure about the correct values to use; 5) having a PC which stops booting and needs major repair or reinstallation of the operating system; 6) recovering from a hard-disk crash; and 7) migrating efficiently to a new PC. Coping with these situations should be routine and straightforward, but in practice such incidents are typically stressful and often waste a considerable amount of productive time.

PAC shares the goals of personal computing—responsiveness, ease of use, and flexibility—with those of AC—simplicity, availability, and security [3].

Personal computing also creates some problems for the implementation of autonomic principles. In particular [2], personal computing users are often, of necessity, system administrators for the equipment they use. Most are amateurs without formal training, who perform system operations infrequently. This reduces their effectiveness and typically requires them to consult with others to resolve difficulties.

This paper presents relevant background and related work before proceeding to discuss the PAC architecture in Section III. These concepts are explored in a proof-of-concept tool that embodies reflex self-healing. We detail these results in Sections IV–VI and conclude with some observations and suggestions for future work.

II. Background and Related Work

A. Autonomic Computing (AC)

IBM introduced the AC initiative in 2001, with the aim to develop self-managing systems [4]. With the growth of the computer industry, with notable examples being highly efficient networking hardware and powerful CPUs, AC is an evolution to cope with rapidly growing complexity of integrating, managing, and operating computing-based systems. Computing systems should be effective [5], should serve a useful purpose when they are first launched, and continue to be useful as conditions change. The realization of AC will result in a significant reduction in system management complexity [6].

The autonomic concept is inspired by the human body's autonomic nervous system [6]. The autonomic nervous system monitors heartbeat, checks blood sugar levels, and keeps the

TABLE I
SERVER VERSUS CLIENT DIFFERENCES

Servers	Personal computers
Hundreds	Thousands
Small set of applications	Open-ended set of applications
Controlled, stable configuration	Less-controlled, dynamic configuration
Fault tolerance through redundancy	Hardware redundancy not affordable
Multiple OS instances through virtualization	One OS instance
Dedicated service processor	All management via main processor
Trained managers	Manager/users
Fixed location	Transportable or mobile
Stable networking environment	Dynamic and intermittent when mobile

body temperature normal without any conscious effort from the human being. There is an important distinction between autonomic activity in the human body and autonomic responses in computer systems. Many of the decisions made by autonomic elements (AEs) in the body are involuntary, whereas AEs in computer systems represent tasks that the system designer has explicitly chosen to delegate to self-contained logic [6].

Upon launching AC, IBM defined four key "self" properties: self-configuring, self-healing, self-optimizing, and self-protecting [6], [7]. In few years the self-x list has grown as research expands bringing about the general term *selfware* or *self-** properties, yet these four initial self-managing properties along with the four enabling properties—self-aware, environment aware (self-situated), self-monitor, and self-adjust—cover the general goal of self-management.

B. PAC

As stated, PAC is AC in a personal computing environment [2]. In some respects, achieving AC within server environments will be an easier task than within personal computing. Servers are likely to have received the level of investment to ensure in-built fault tolerance and include extensive redundancy—including facilities such as hot swapping [8]. Personal devices are often machines built on the faster, cheaper, and smaller philosophy with limited, if any, redundancy and as a result the self-healing logic has fewer and less effective alternatives. Servers are also likely to have a user base of highly skilled teams, whereas personal devices are often in the hands of nontechnical users who often also act as the administrator. Other considerations are required for personal computing such as flexibility of location (e.g., laptops) and of hardware (e.g., palm devices) and software configuration that complicate further the goal of achieving AC [2], [5]. Specifically in terms of PCs used for client versus server applications, Table I highlights typical differences in the way an organization uses them, resulting in different management requirements for the two domains. As such it is not just the case that PAC is AC implemented on a PC—there are many other considerations related to the way these PCs are used. Added to this, personal computing is not just the PC—consider the pervasiveness of all manners of personal devices with such future directions toward wearable computing and smart homes.

The following are the common examples of existing autonomic capabilities within personal computing.

Self-configuring: Microsoft Windows XP has an automatic update function. It updates itself to catch updated or newly released component(s) [2].

Self-healing: Windows XP Professional provides a checkpoint function to backup the system and to permit the user to recover if the system has crashed.

Self-optimizing: Microsoft Windows XP Professional now optimizes the user interface based on the way the system is used. For instance, it attempts to keep the desktop clean and uncluttered by removing items not recently used. Because of the nature of personal computing, the user is asked to confirm that these changes take place [2].

Self-protecting: An example of a protection mechanism is encryption. Windows XP is built with an encryption capability that allows directories to be encrypted. Microsoft Internet Explorer is embedded with security protocols such as SSL and TSL (a downside is encryption makes self-healing harder to achieve on different technologies). Norton's Antivirus (Symantec Corporation) software automatically scans all emails to check if they contain any virus. Microsoft Excel prompts an alert if the user opens a spreadsheet containing a macro which may have a virus.

C. Peer-to-Peer (P2P)

P2P is a paradigm in which each workstation on a network can dynamically serve in any role, for example, as client or server [9]. This differs from the Client/Server architecture, in which a server is a dedicated computer to serve client requests. The server machine is usually always available so that the clients can connect to it at any time, while in a P2P network there may not be such availability guarantees. P2P is not a new concept; IP routing is a classic example. P2P computing offers the promise for an organization of cost-efficient sharing of computer resources, improving network performance, and increasing overall productivity.

Peer frameworks are becoming mainstream, for instance, JXTA [10] and Microsoft's peer framework update [11]. The P2P paradigm is also a key in ambitious future plans for virtual file servers that would be accessible to a hundred thousand [12] or even link billions [13] of individual computers. It offers the flexibility required for achieving autonomic personal computing, for instance, it makes available additional resources to a (redundancy-poor) PC for self-healing.

The first P2P systems had computers connected together as a workgroup and configured for the sharing of resources such as files and printers. In particular, the computers were located near each other physically and ran on the same networking protocols. Today, computers are connected together over the Internet. Computers (including handheld devices) can join the network from anywhere with little effort.

P2P architectures enable computers to dynamically share services and resources directly between one another. P2P participants range from a large server to a handheld device. Resources and services include the exchange of information, processing cycles, cache storage, and disk storage, as well as

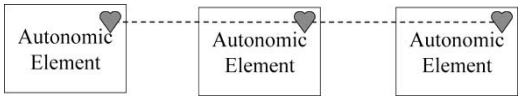

Fig. 1. Heart-beat monitoring ("I am alive" signals).

higher level services [14]. An application of P2P technologies is for reclaiming unused computing cycles on desktop computers and harnessing them into a virtual supercomputer [15]. In this scenario, a large job can be broken into small parts and run on separate machines in parallel. At the same time, it reduces the load on servers, allowing them to perform specialized services more effectively. In the P2P-enabled distributed computing model, a managing server is configured to send different pieces of one computing job to a set of peers, who then distribute it on to second-tier peers, then third-tier peers, and so on. P2P can also facilitate geographically dispersed collaborative computing. As with file sharing, collaboration can decrease network traffic by eliminating e-mail and can decrease server storage needs by storing files locally. P2P computing also allows domains to collaborate using intelligent agents [14]. In terms of self-protection, sharing virus alert information is an example.

D. Heartbeat and Beacon Monitoring

System management is typically based on events that are generated by a process when fault or problem conditions occur in that process. In embedded systems, the opposite is typically the case—a management action is taken when an independent process detects that an expected event has not occurred. An example is the fault tolerant mechanism of a heartbeat monitor (HBM); through a combination of the hardware (the timer) and software (the heartbeat generator) an "I am alive" signal is generated periodically to indicate all is well [16]. The absence of this signal indicates a fault or problem. Some embedded processors have a hardware timer which, if not periodically reset by software, causes a reset/restart. This allows a particularly blunt, though effective, recovery from a software hang. It may cause a perceptible outage in the system's function and may even cause lost data.

The approach of independent process monitoring offers the advantage that through continuous monitoring problem determination becomes a proactive rather than a reactive process.

The HBM has now been adopted in system management architectures. In Grid Computing, the Open Grid Services Architecture (OGSA) has a facility referred to as the Globus HBM which is designed to detect and report whether registered processes are still alive or not [17], by detecting the absence of an heartbeat. The HBM may be considered a specific type of environment awareness, since from a system perspective these heartbeats provide awareness of the individual functioning elements [5] (Fig. 1).

Deep Space 1 (DS1) [18], [19] was launched in July 1998 by the National Aeronautics and Space Administration (NASA). The beacon monitor was 1 of the 12 new technologies used in DS1. Its goal was to decreasing the total volume of engineering telemetry, through reducing the frequency of use of the downlink and the volume of data received per pass [19]. With beacon monitoring, the spacecraft assesses its own health and transmits one of four subcarrier frequency tones to inform the ground how urgent it is to track the spacecraft for telemetry [20]. Table II summarizes the tone definitions.

TABLE II
BEACON TONE

Tone	Description
Nominal	All functions as expected. No need to downlink.
Interesting	Interesting – nonurgent event. Establish communications when convenient.
Important	Communications need to take place within timeframe or else state could deteriorate.
Urgent	Emergency. A critical component has failed. Cannot recover autonomously and intervention is necessary immediately.
No tone	Beacon mode is not operating.

The two primary flight software innovations implemented through the beacon monitor were onboard engineering data summarization and beacon tone selection [19]. The tone selector module maps fault protection messages to beacon tone states. Transforms and adaptive alarm thresholds are the components to create top-level summary statistics, episode data, low-resolution "snapshot" telemetry, and user-defined data. These two components aim to minimize the number of false alarms.

These approaches influence our architecture for PAC, and we adopt the taxonomy of urgency that they define.

III. PAC ARCHITECTURE

The goal of an AC environment is self-management. The four self-managing properties are self-configuring, self-healing, self-protecting, and self-optimizing [6], [7], together with the attributes of self-awareness, environment awareness (self-situated), self-monitoring, and self-adjusting. Self-healing is concerned with ensuring effective monitoring, diagnosis, and recovery when a fault occurs, without human interaction. To achieve the self-healing objective, a system must be self-aware and environment aware. A system would be aware of its internal state as well as the external operating conditions. The system would have knowledge of its available resources, components, desired performance characteristics, current status, and status of interconnections with other systems [7].

In this research, P2P approach is utilized due to the dynamic realities of personal computing. The peers form a "neighborhood watch" scheme so as to assist with self-management and look out for each others' health. Peers take on the responsibility of low-overhead monitoring of other peers and can help initiate self-healing activities when the failing system cannot.

A. Reflex and Healing

Reflexes and healing are complementary strategies inspired by biological systems [21]. Animals have a reflex system, where the nerve pathways enable rapid response to pain. Reflexes cause a rapid involuntary built-in preplanned motion, such as when a

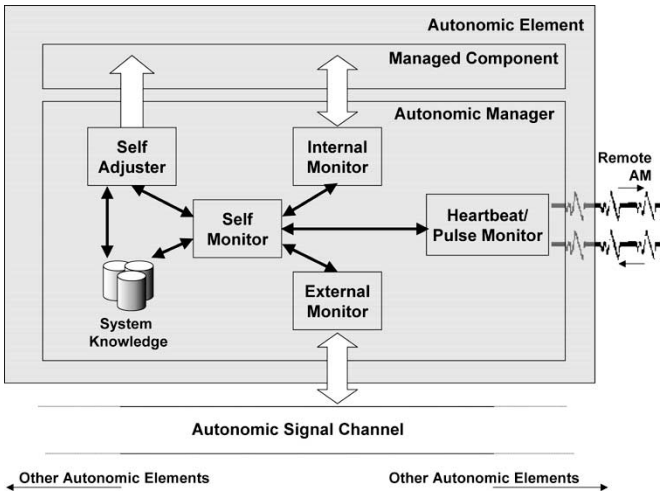

Fig. 2. Architecture of a PAC element.

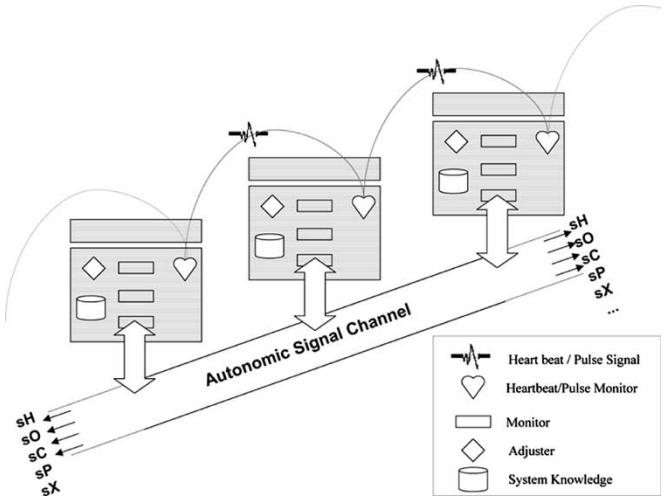

Fig. 3. Autonomic environment.

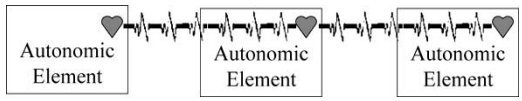

Fig. 4. Pulse monitoring ("I am/am not healthy" signals).

sharp object is touched. The effect is that the system reconfigures itself, moving away from the danger to keep the component functioning.

The body will heal itself on a much longer timescale. Resources from one part of the system are redirected to rebuild the injured body part, including repair of the reflex response network. While this cannot help in the real-time response, directly after an event, it can prepare the system for the next event. In addition, it can readjust the system for operation with a reduced set of resources [21].

These complementary strategies resolve a dilemma: how can a system react quickly to limit damage and also perform the complex reconfiguration and regeneration to recover from it? The answer is to provide different mechanisms, which function on different time scales, each optimized for part of the task.

B. PAC Element

Achieving high usability and security for personal systems requires rapid accurate responses to changing circumstances. The PAC architecture incorporates a mechanism equivalent to the biological reflex reactions to alert members of the peer group to situations requiring urgent attention. In general, a system will have to reconfigure itself to avoid a detected threat, while maintaining its operation as far as possible. This may result in the system operating with a reduced set of resources [21]. Like the human body, a system can then address the problem causing the reaction with less urgency; this may involve some damage repair.

Fig. 2 shows an abstract view of a system architecture to support this model [5], [22]. This is similar in nature to the architecture proposed in the IBM blueprint where an autonomic manager (AM) consists of monitor, analyze, plan, and execute along with knowledge (MAPE-K) components [6].

An AE is made up of a managed component and an AM. The self-monitor actively observes the state of the component and its external environment, drawing conclusions using information in the system knowledge base. If necessary, this can lead to adjustments to the managed component. An additional feature is the use of an extended version of a HBM, called a pulse monitor (PBM) [22], to summarize the state of the managed component for other connected AEs. Essentially it provides an indication of the health of the managed component (self-awareness) or external environment (environment awareness) as viewed by that manager, with the absence of a signal (heartbeat) indicating a specific problem with the manger itself. The signal itself, like a human pulse, can provide additional information to further explain the state of the element and trigger reflex actions [23]. Pulse monitoring is discussed in more detail in the next section.

Key within this AE is the ability to provide self-management through a combination of a control loop (self-monitor and self-adjuster) and the system knowledge.

Fig. 3 depicts how the AEs are logically connected. The artifacts within an AE (Fig. 2) and AEs within a system (Fig. 3) communicate, for instance, via a logical communications channel using asynchronous communicating techniques, like a message bus [6]. The logical difference between the pulse signal and general event messages has been highlighted in Figs. 2 and 3, since essentially the pulse provides the mechanism for a reflex reaction whereas the general event messages under fault conditions form part of the slower healing process-root cause analysis from the event stream.

C. Pulse Monitoring

A hybrid approach for the autonomic environment [23] is to use the urgency concept of the beacon monitor to turn the HBM into a pulse monitor—so instead of just checking the presence of a beat, the rate is also measured (Fig. 4).

The concept of the pulse monitor is based on extending the HBM construct. The HBM itself provides a means to ensure a

vital process may be safeguarded. The lack of a heartbeat will alert the designated remote HBM that the process has died (or indeed the communications themselves have failed). This relatively instant alert to the fact a process is no longer functioning enables immediate actions such as restarting the process and as such minimizing disruption.

Essentially, the HBM provides a vital construct, without which the system is relying on another process noticing that the process has died with no guarantee on how much time will have lapsed before this occurs, if at all.

Yet, vital as it is, essentially the HBM only informs if a process is alive or dead (assuming communications are working), not the processes' actual health or state of being. Taking the biological analogy, the rate of the heartbeat indicates the current conditions within which the biological system is operating and determines strategies for components within the system (for example, increased heart rate may indicate increased blood flow through the body due to the individual changing from walking to running). Choosing the right rate for the pulses is key, since if they are too frequent, resources are consumed unnecessarily; if too infrequent, the latency of detection is too long and damage may propagate.

An important point to note from the HBM, and also from the Beacon Monitor, is the minimization of data sent, essentially only a signal is transmitted. Any move towards sending more information must not compromise this reflex reaction. As such, the tone or the beat must contain within it the urgency level.

This effectively may be used to provide a reflex reaction within the autonomic environment and adds the dual approach of reflex and healing, sharing responsibility for self-monitoring and environment monitoring among peers.

The pulse monitor has been recommended as an extension of the HBM for Grid computing [22], as a construct within an AM [5], [23] and a reflex mechanism within a telecommunications fault management architecture [24]. The research reported here is utilising the pulse monitor as a construct within the PAC architecture which specifically demonstrates self-healing. The demonstration self-healing tool operates in a P2P mode without any additional environment on top of the Windows OS.

D. Reflex Self-Awareness

The PBM monitor may be used as a means to quickly (a reflex reaction) communicate the state of the individual *self* to other AMs (peers), i.e., through internal monitoring of a managed component vital signs (e.g., failing processes) give an indicator to the state of health and used to bring about a change in pulse being emitted from the *self*. This information could then be used in various ways, for instance, to enable remote recovery strategies or avoid allocation of workload to that element experiencing difficulties.

E. Reflex Environment Awareness

The PBM may also be used for environment awareness. In this scenario, the pulse becomes a shared value of environment health as opposed to an individual's health value (Fig. 5). From Fig. 2 it is clear each machine (the managed component) must have its own internal monitoring but it is also clear that each AM needs to monitor the external environment (at least two control loops per AE). In a local P2P network in many circumstances it may be assumed that the environment the peers are all monitoring is the same environment, resulting in a waste of resources through duplication [Fig. 5(a)]. To reduce this duplication, an individual peer or group of peers (e.g., the least busy PC or assigned PCs) may take on the monitoring role for the group [Fig. 5(b)]. Again taking into account the required flexibility of personal computing, other peers must be ready to take on monitoring responsibilities should circumstances change (a laptop is removed from the peer group by the user departing the office). The role of the PBM in these circumstances is to provide a reflex reaction from the environment monitoring (external monitoring) peer to the nonenvironment monitoring peers to alert them to an environment situation.

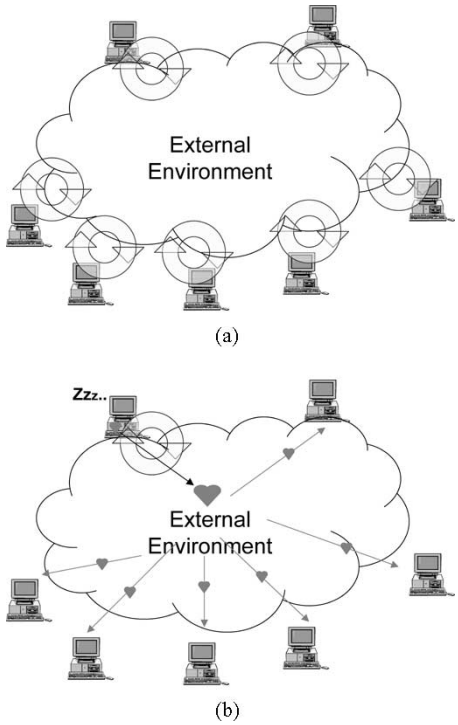

Fig. 5. Reflex environment awareness.

The PBM utilized for environment awareness should be considered in addition to the self-awareness pulse mechanism, i.e., utilized to create a shared dynamic group environment awareness as well as individual self-protection.

The following sections detail a proof of concept PAC self-healing tool based on these principles.

IV. Reflex Self-Healing Tool Design

Self-healing is an emerging research discipline in itself [25]. Personal computing with its flexible nature and diverse user base [2] often makes self-healing harder to achieve.

The assumption behind this self-healing tool used to demonstrate the concepts and architecture discussed in the previous sections is that dying/hanging processes on the PC are signs or

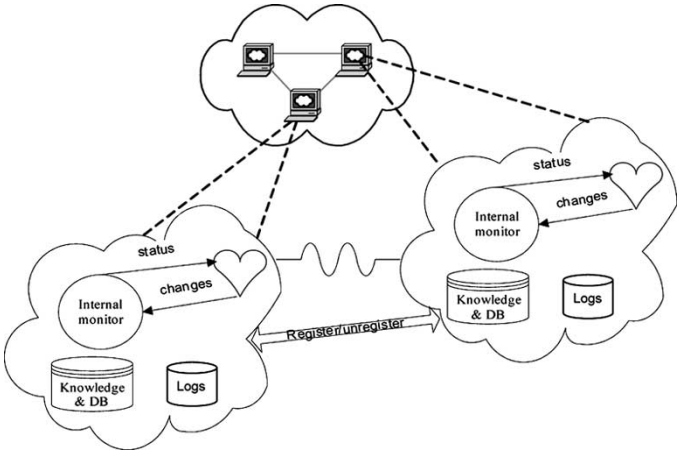

Fig. 6. P2P pulse monitoring.

indicators of the health of that PC. These vital signs may indicate that the PC is becoming unstable and is possibly in imminent danger of hanging or becoming unreliable for current processes running on that machine. As well as restarting the detected hung process(es) the peers are notified of the situation via a change in pulse.

This is particularly useful in situations where the PC is unattended (for example, running a web server) and the user may be notified via a peer PC that the machine is in difficulty. It is also useful when machines in the peer group are sharing work load, for example, via Harmony PC grid services [26]; a peer is notified in advance of the immanent danger and can recover data and reallocate work to another peer. Such an approach is more proactive than responding once the machine (managed component) has hung, and as such offers fuller potential for autonomic capabilities.

The underlying functionality of the tool is a heart-beat monitor; if a process hangs it should be restarted and the pulse monitor takes note. Upon several processes hanging or the same process repeatedly hanging within specified timeframes, a change occurs in the monitor's perception of how healthy the machine is, and as such brings about a change in the pulse being broadcast to an assigned peer from that PC.

Since the tool operates in a P2P mode it also takes responsibility to monitor its neighbors; as such other PCs (peers) will register with it and it will monitor their pulse. Since PBM is P2P, all hosts have equivalent capabilities and responsibilities. They are monitoring each other with minimal human interaction. A host retains its autonomy and can choose either to register or not register with other hosts. Two hosts become neighbors after they register to each other. There is no logical limit on how many neighbors that a host can register with. Hosts send pulses to each other only when they are connected. If a host becomes unavailable, an alternate registration may be made.

Fig. 6 depicts an overview of the pulse monitor construct. An internal monitor inside a host takes care of monitoring its health condition. This health condition will be represented by a pulse. Each host is able to send its pulse to a peer via an external monitor. The "knowledge & database" stores the pulse

TABLE III
P2P PULSE VALUE

Urgency level	Description	Pulse change trigger (adaptable)
0	Nominal	no failed process
1	Interesting	1 failed process
2	Important	2 failed processes
3	Urgent	3 or more failed processes
—	No pulse	Pulse monitor, or comms has failed

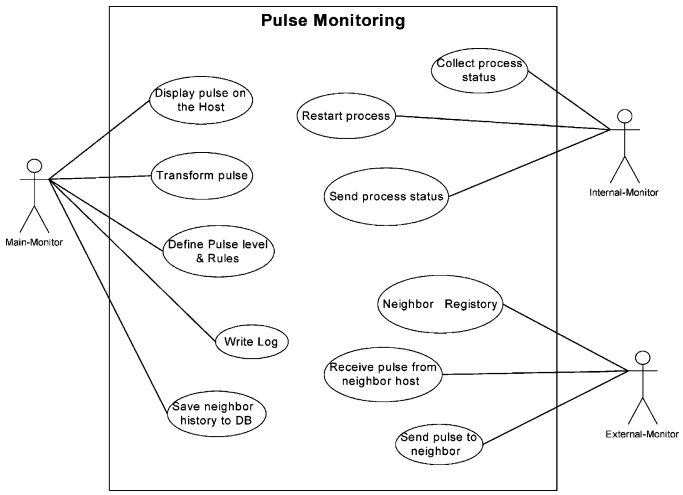

Fig. 7. P2P pulse monitoring use-case diagram.

level and rules (i.e., predefined knowledge) which may adapt over time, the monitoring logs, and the history of neighbor hosts. A computer system is different from a biological system; firstly, since the human biology reflex reaction is involuntary while the reflex reaction decision making in computer systems is delegated to the system by the user (through a set of rules or policies) and secondly, the extent to which the reaction can be reconfigured. Since an argument could be made, that in a biological system the reflex reaction is wired-in through genetic adaptation over time, the important difference to focus on is the extent of the ability to reconfigure the reflex reaction. For example, rules such as the "pulse sending interval" and "change pulse after three failed process" are reconfigurable.

The host sends the degree of urgency to the peer's pulse external monitor instead of just a beat. The urgency level is transformed based on the number of failed processes (Table III).

The amount of processes required to cause a change in the pulse is adaptable and need not necessarily remain at the values depicted in Table III, as is the time window for qualifying failing processes. The connection between two hosts is established using the User Datagram Protocol (UDP) since the pulse essentially only transmits small size messages.

Fig. 7 is a UML use-case diagram that summarizes the functionality of the pulse monitor API. The pulse monitor scans the host periodically to check its health condition; it transforms the health condition to a pulse value and will send it to connecting assigned neighbor. If a process is found to have failed, the tool will try to restart that process.

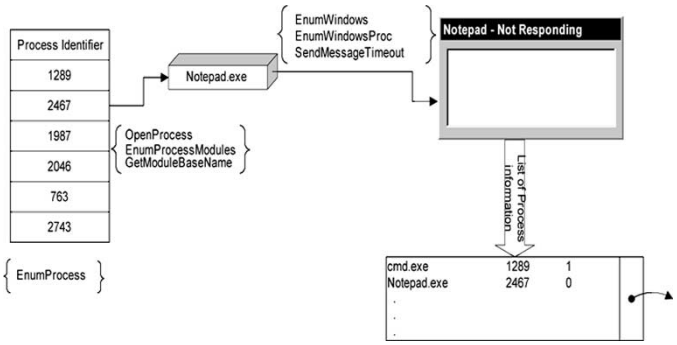

Fig. 8. Obtaining process information.

V. Reflex Self-Healing Tool Implementation

This section looks at the implementation of the proof of concept. Parts of this implementation are unavoidably OS and OS version specific, but the hierarchical implementation keeps much of the prototype generic.

A. Health Monitor Implementation

The pulse monitor is developed in Java. The health monitor operates under the Microsoft Windows environment using the running processes (through the active process list) as vital health signs. Since this health component is OS specific it was developed in C with the Windows SDK.

In the Windows environment, as in other OSes, applications consist of executable files and dynamic link libraries (DLLs) [27]. A running application is known as a process; a process consists of one or more threads, where a thread is the basic unit to which the operating system allocates processor time. Each process is assigned an identifier which is valid until the process terminates. A module is an executable file or DLL. Each process consists of one or more modules [27].

Fig. 8 illustrates how the process information list is obtained from the Windows platform. The performance monitoring components in the Windows Platform Software Development Kit has the technologies to deal with process, thread, module, heap, processor, memory, and event. The Process Status Helper (in psapi.dll) provides an interface to obtain information about processes [27].

Within the Windows system, the EnumProcesses function retrieves all running process identifiers, the OpenProcess function opens an existing process object to obtain the handle of a process, and the EnumProcessModules function retrieves a handle for each module of a process, while the GetModuleBaseName function retrieves the name of a module. A list of running processes with their identifiers and names can be obtained by using these functions; however, the list does not have the processes status. To establish if a process is running normally or if it has hung, it is necessary to first obtain the window handle of that process and then send a message to the window to see if it can respond or not. The EnumWindows function enumerates all top-level windows and as such has to be called with the EnumWindowsProc function. The EnumWindowsProc function is an application defined callback function. It receives top-level window handles and is a placeholder for the application defined function. The window handle associated with a process is then passed to the SendMessageTimeout function to check if the window is responding or not. It returns without waiting for the time-out period to elapse if the window appears to not respond or has hung.

The health-monitor will terminate a process if the process has failed but cannot be recovered (or restarted). The Window's function TerminateProcess terminates a process and all of its threads. It stops execution of all threads within the process and requests cancellation of all pending I/O.

The constant monitoring is essentially the control loop in this situation; self-monitoring of the processes and reaction (self-adjusting) if necessary (Fig. 2).

B. Health Monitor and Pulse Monitor Interfacing

The Java Native Interface (JNI) [28] is used to interface between the Java based pulse monitor and the C coded health monitor. JNI defines a standard naming and calling convention so the Java virtual machine (JVM) can locate and invoke native methods. With JNI, native methods can create, update, and inspect Java objects; Java can pass any primitive data types or objects as parameters to native methods; native methods can return primitive data types or objects back to the Java environment; Java instance or class methods can be called from within native methods; native methods can catch and throw Java exceptions.

The interface could have been developed in COM or J/Direct instead of the JNI approach; however, these provide solutions which are even more OS specific [29]. With the approach used, in order for the health monitor to run on different platforms such as UNIX, a modification of the collect process information methods in the C program is required.

Since the process list (the source of health indicators) is dynamic, an array is not flexible enough to store the list. A Vector class is used to hold the process information. The vector class in Java is designed to store heterogeneous collections of objects thus providing methods for working with dynamic arrays of varied element types. Three vector variables are declared in the Java program to hold process information: process name, process identifier, and process status.

C. Pulse Monitor Implementation

The External-Monitor (Figs. 2 and 9) provides the communications function with other peers, using UDP sockets (see Fig. 10). UDP is described as unreliable, connectionless, and message oriented [30], yet is good for sending short messages like those required for the pulse monitoring application, where all messages are less than 100 B. A socket is a handle for a communications link over the network to another application [30].

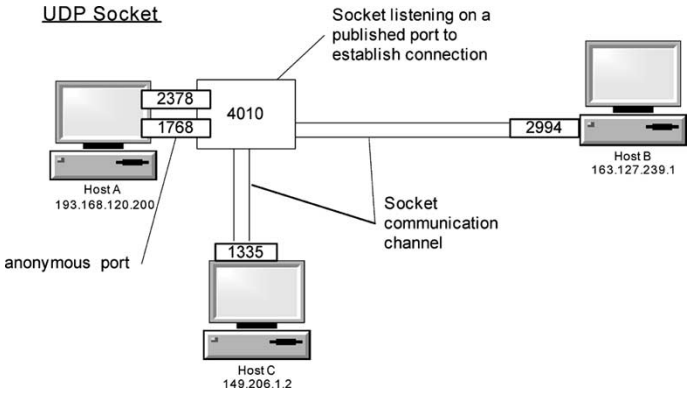

Fig. 9. Socket communication.

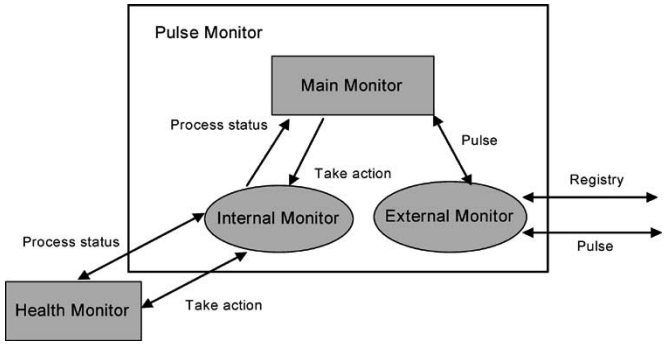

Fig. 10. Pulse monitor within the autonomic element.

A packet is a self-contained message that includes information about the sender, the length of the message, and the message itself. The send function sends out a datagram packet to a destination address. The receive function blocks until a datagram packet is received. It waits for a packet forever unless a time-out is enabled.

The Java code to communicate using UDP carries out the following: creates an appropriately addressed datagram to send; sets up a socket to send and receive datagrams for a particular application; inserts datagrams into a socket for transmission; waits to receive datagrams from a socket; decodes received datagrams to extract the message, its recipient, and other metainformation. The DatagramSocket class provides a function to create a socket object and communicate packets [28]. A datagram socket is the sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed. The DatagramPacket class represents a datagram packet, which are used to implement a connectionless packet delivery service. Each message is routed from one machine to another based solely on information contained within that packet.

Before two hosts can send the pulse to each other, they first have to register to each other. When the External-Monitor starts, it immediately connects to its registered neighbor. The External-Monitor disconnects from all connecting neighbors when it ends. Unregistering from a neighbor will remove that host from its neighbor list and they will no longer send the pulse to each other.

UDP is an unacknowledged service protocol. The use of only one port to serve all messages may overload the port and hence increase the probability of losing a message. There are six UDP sockets created on different ports to wait for incoming messages:

1) to register to it, port 4001;
2) to unregister from it, port 4002;
3) waiting neighbors connecting to it, port 4003;
4) waiting neighbors disconnecting from it, port 4004;
5) neighbors sending pulse to it, each host defines its own port number;
6) waiting neighbors to check if the host is still on or not, port 2222.

Time-out is not enabled on these sockets because they have to wait for incoming messages forever. When PBM receives a message, it then calls the corresponding function and replies with an acknowledgment to the sender. The advantage of using multiple ports is that there is less decoding/dispatching to do. To send a message, a separate socket port is opened. The time-out is enabled to wait for a reply to ensure the message is delivered. When finished, this socket will close (i.e., providing the functionality of any acknowledgment-based protocol like TCP with the lighter implementation of UDP.)

The health tool and pulse monitor are made up of four components: Main-Monitor, Internal-Monitor, External-Monitor, and Health-Monitor (Fig. 9): All components have to be executed synchronously since multiple jobs are required to be carried out at the same time, and thus the tool utilizes Java's built-in support for threads to achieve multitasking.

The Internal-Monitor is responsible for the monitoring of processes running on the machine (the managed component). The Health-Monitor is responsible for obtaining process information and cleaning up and rebooting failed processes. The External-Monitor is responsible for the communication between its neighbors (sending and receiving pulses). The Main-Monitor is the coordinator and decision maker in terms of instigating a change in the emitted pulse level (Fig. 9).

VI. Reflex Self-Healing Tool Results

A. Self-Healing Scenario

Fig. 11(a)–(f) shows screenshots from the reflex self-healing tool running on Windows XP. For clarity, parts of the shots are enlarged [Fig. 11(b)–(f)]. The figure depicts a scenario where several process have hung, causing the changes in the pulse from nominal to interesting to urgent and back to nominal as the self-healing tool successfully cleans up and restarts the processes. In this scenario, the user was viewing a pdf file on the IEEE web site [involving Netscape web browser, Acrobat reader, and touch pad driver (appoint.exe)] and had attempted (impatiently) to scroll ahead before the document had fully downloaded. This action had resulted in Netscape/Acrobat hanging. With the self-healing tool running it automatically detects that the processes have hung and restarts them. The second restarting of Netscape in plate f arose as this user utilizes Netscape Mail which had

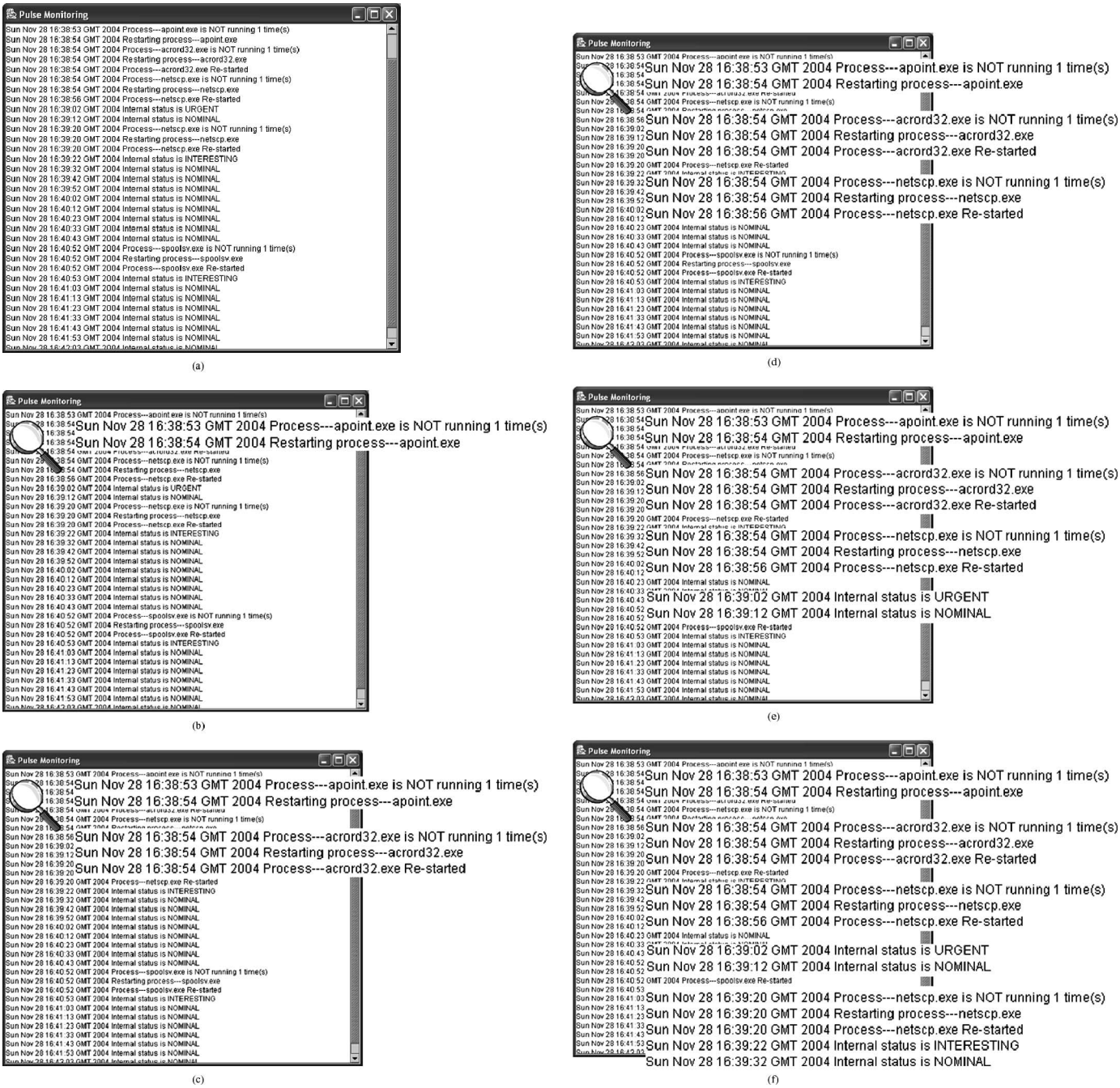

Fig. 11. Developing scenario within the self-healing tool.

thousands of emails in the inbox, causing a delay in restarting/reloading the Netscape application.

The tool works best when the user is not directly involved and the alert to another peer can bring about a useful interaction (either a self-healing strategy or alerting a human to the problem), for instance, when running a personal web server such as Apache which can be restarted once hung. The self-healing tool offers a much improved solution—instead of waiting for an administrator to be informed by a user who has noticed that the web site is down and takes time to report it, the server is rebooted automatically when it hangs; if the difficulty continues and the rebooting is not successful, the administrator may be alerted by the tool through the pulse or the alert from the pulse can trigger a failover to another system. The scenario in Fig. 11, with interactive applications, was chosen to highlight the challenges in PAC as the user is very much in the loop. In the scenario the user had just noticed that Acrobat had hung. The side effects of the tool successfully clearing up the multiple hung processes and restarting them was that these applications suddenly disappeared from the screen and then restarted creating unexpected behaviour, which may be considered to break with computer–human interaction (CHI) guidelines.

Another challenge is although the applications have restarted for the user, their state has changed: Netscape and Acrobat do

not show the web page before the restart. The problem here is that there is no standard means to capture the process state when it is terminated and to restart the process with that state—effectively a process is started from fresh with any previous state lost unless the process' application itself handles this by internally checkpointing.

B. Need for Standards and Microrebooting

The scenario has highlighted the need for standards, for autonomic signals, communications, and for checkpoint to take place not only at the level of AM to processes running on the managed component, but also at the level of AM to AM. Allowing standard "autonomic signal" routes into processes would raise security issues if handled at the application level—yet this will need to be a part of the self-protection autonomic property. On the other hand, Windows and Linux processes have a standard kill function and this is not considered a security issue. Basically, the infrastructure being considered is OS-like and therefore trusted.

This implies all processes and OSes need to be designed with autonomicity and self-managing capabilities in mind, i.e.,capable of taking direction from the external environment. This not only raises issues of standards to achieve this but raises questions as to whether the current design and development tools meet the needs for developing process of this type.

Recursive microrebooting is a promising nonexpensive approach for self-healing [31], which fits directly with the concepts in this PAC reflex and healing research. It seeks to create software components (at various levels of granularity) that are "crash only," that is, self-healing is to reboot them, and to do so at as low a level as feasible so the user does not see the physical impact of an application restarting.

VII. Conclusion

Overall, AC is intended to improve the general usability and manageability of computing systems and so, in principle, will benefit all computer users in due course. Since for the majority of users access to computing is through personal devices, autonomic research in this area should have a significant impact. In the longer term, the work is of direct relevance to emerging important areas, such as utility/grid and ubiquitous computing, which require systems to self-manage to fulfil their potential. These will provide broad support for eScience, eGovernment, eHealth, and eBusiness applications which for the foreseeable future will be accessed by the majority of users through personal computing.

The research in this paper explores a novel computing structure for the distributed realization of autonomic behavior in personal and embedded systems. AC behavior is commonly implemented in an AM, most often a component of a managed system, but sometimes separated from that managed system in a management server. The flow of monitoring and management is most often hierarchical. This flow is appropriate for systems of static or near-static structure, with adequate resources to devote to the AM and to communications between it and the managed system.

The contribution in this research is a way for AMs to share data and management decisions in a nonhierarchical way, even in an *ad hoc* manner. It permits one AM to monitor the health of other AMs without necessarily controlling them and even to reach decisions based on consensus. It opens up opportunities for the collaboration of AMs in a way that is less rigid than in current AC architectures. This style of AC is much more appropriate to personal and embedded computing because it supports the dynamic self-centered style exhibited by this type of computing.

The reflex reaction notion is based on the need for a system to respond more quickly than it can respond after detailed analysis and planning of a comprehensive response. Often, the first part of a comprehensive response can be preplanned: in the case of virus infections the first step is to disconnect from the network. It has been observed that failures sometimes occur in recovery from virus infections, where the OS is reinstalled and is reinfected during the installation process itself. Recovery actions when connected to a network hosting a virulent virus infection have to be performed in less time than it takes to get infected. Also, since comprehensive analysis in preparation for response may involve contacting neighbors and servers and waiting in their queues, the time to do this comprehensive analysis may be long. To address this requires research into the dynamics of infection, analysis, and recovery so as to supply bounds on the amount of time that a system can take before it starts to take action.

Further, autonomic options could involve learning from past behavior; for example, the ability to spot a process running intermittently or unstably. The process may have a history of failing after running for a certain period of time or after attaining some state. In this case, the process (the application) may need reconfiguration or reinstallation in order to run reliably. The goal is to provide options for self-configuring and self-optimizing and in so doing to prevent the system from degrading further; thus providing proactive self-protection and self-healing.

References

[1] P. Horn "Autonomic computing: IBM perspective on the state of information technology," IBM, Armonk, NY, Oct. 2001.

[2] D. F. Bantz, C. Bisdikian, D. Challener, J. P. Karidis, S. Mastrianni, A. Mohindra, D. G. Shea, and M. Vanover, "Autonomic personal computing," *IBM Syst. J.*, vol. 42, no. 1, pp. 165–176, 2003.

[3] D. F. Bantz and D. Frank, "Challenges in autonomic personal computing, with some new results in automatic configuration management," in *Proc. IEEE INDIN; AUCOPA Workshop*, Banff, AB, Canada, Aug. 22–23, 2003, pp. 451–456.

[4] A. G. Ganek and T. A. Corbi, "The dawning of the autonomic computing era," *IBM Syst. J.*, vol. 42, no. 1, 2003.

[5] R. Sterritt and D. W. Bustard, "Towards an autonomic computing environment," in *1st Int. Workshop Autonomic Computing System in IEEE Workshop Proc. 14th Int. Conf. Database and Expert Systems Applications (DEXA'2003)*, Sep. 1–5, 2003, pp. 694–698.

[6] IBM, "An architectural blueprint for autonomic computing," Apr. 2003.

[7] R. Sterritt and D. Bustard, "Autonomic computing—A means of achieving dependability?," in *Proc. IEEE Int. Conf. Engineering of Computer Based System (ECBS'03)*, Huntsville, AL, USA, Apr. 7–11, 2003, pp. 247–251.

[8] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis, "Enabling autonomic behavior in systems software with hot swapping," *IBM Syst. J.*, vol. 42, no. 1, pp. 60–76, 2003.

[9] What is P2P. [Online]. Available: http://compnetworking.about.com/library/weekly/aa093000a.htm

[10] Project JTXA (2004). [Online]. Available: http://www.jxta.org/

[11] Microsoft. (2004). *Peer-to-Peer Infrasture*. [Online]. Available: http://msdn.microsoft. com/library/default.asp?url=/library/en-us/p2psdk/p2p/portal.asp
[12] Microsoft Research. (2004). *Farsite Project*. [Online]. Available: http://research. microsoft.com/sn/Farsite/
[13] UC-Berkeley. (2004). *OceanStore Project*. [Online]. Available: http://oceanstore. cs.berkeley.edu/
[14] Peer-to-Peer Computing is Good Business. [Online]. Available: http://www.intel.com/eBusiness/products/peertopeer/ar010102.htm
[15] Peer-to-Peer Architecture. [Online]. Available: http://80211-planet. webopedia.com/TERM/p/peer_to_peer_architecture.html
[16] P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. V. Laszewski, "A fault detection service for wide area distributed computations," in *Proc. 7th IEEE Symp. High Performance Distributed Computing*, Jul. 28–31, 1998, pp. 268–278.
[17] The Globus Heartbeat Monitor Specification v1.0. [Online]. Available: http://www-fp.globus.org/hbm/heartbeat_spec.html
[18] D. DeCoste, S. G. Finley, H. B. Hotz, G. E. Lanyi, A. P. Schlutsmeyer, R. L. Sherwood, M. K. Sue, J. Szijjarto, and E. J. Wyatt, "Beacon monitor operations experiment DS1 technology validation report," Jet Propulsion Lab, California Inst. Technol., Pasadena, CA, 2000.
[19] E. J. Wyatt, M. Foster, A. Schlutsmeyer, R. Sherwood, and M. K. Sue, "An overview of the beacon monitor operations technology," presented at the Int. Symp. Artificial Intelligence, Robotics, and Automation in Space, Tokyo, Japan, 1997.
[20] E. J. Wyatt, H. Hotz, R. Sherwood, J. Szijjarto, and M. Sue, "Beacon monitor operations on the deep space ONE mission," presented at the 5th Int. Symp. AI, Robotics and Automation in Space, Tokyo, Japan, 1998.
[21] T. Bapty, S. Neema, S. Nordstorm, S. Shetty, D. Vashishtha, J. Overdorf, and P. Sheldon, "Modeling and generation tools for large-scale, real-time embedded systems," in *10th IEEE Int. Conf. Workshop Engineering of Computer-Based Systems*, Huntsville, AL, Apr. 7–10, 2003, pp. 11–16.
[22] R. Sterritt, "Pulse monitoring: Extending the health-check for the autonomic GRID," in *IEEE Workshop Autonomic Computing Principles and Architectures (AUCOPA'2003)*, *in Proc. IEEE Int. Conf. Industrial Informatics (INDIN 2003)* Banff, AB, Canada, Aug. 22–23, 2003, pp. 433–440.
[23] ——, "Towards autonomic computing: Effective event management," in *Proc. 27th Annu. IEEE/NASA Software Engineering Workshop*, Greenbelt, MD, Dec. 2002, pp. 40–47.
[24] R. Sterritt, D. Gunning, A. Meban, and P. Henning, "Exploring autonomic options in an unified fault management architecture through reflex reactions via pulse monitoring," in *IEEE Workshop Engineering of Autonomic Systems (EASe 2004) in Proc. 11th Ann. IEEE Int. Conf. Workshop Engineering of Computer Based Systems (ECBS 2004)*, Brno: Czech Republic, May 24–27, 2004, pp. 449–455.
[25] D. Garlan, J. Kramer, and A. Wolf, Eds. Proc. Workshop on Self-Healing *Systems (WOSS'02),* Charleston, SC. Nov. 2002. New York: ACM Press
[26] V. K. Naik, S. Sivasubramanian, and D. F. Bantz, "Harmony: A desktop grid for delivering enterprise computations," presented at the Proc. 4th Int. Workshop on Grid Computing (Grid 2003), Phoeniz, Arizona, Nov. 2003.
[27] Microsoft Platform SDK Documentation. [Online]. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ perfmon/base/about_performance_monitoring.asp.
[28] Sun Microsystems, Inc., Enterprise Services, Jan. 1999, Revision B.1. *Java Programming Language*.
[29] M. Pietraszak, "Using J/Direct to Call the Win 32 API from Java," [Online]. Available: http://www.microsoft.com/mind/0198/default.asp, 1998.
[30] J. L. Weber and M. Wutka, *Using Java 2 Platform*. Indianapolis, IN: Que, 1999.
[31] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot-A technique for cheap recovery," in *Proc. 6th Symp. Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.
[32] R. Sterritt and S. Chung, "Personal autonomic computing self-healing tool," in *Proc. IEEE Workshop Engineering of Autonomic Systems (EASe 2004) at 11th Annu. IEEE Int. Conf. and Workshop Engineering of Computer Based Systems (ECBS 2004)*, Brno, Czech Republic, May 24–27, 2004, pp. 513–520.
[33] R. Sterritt and D. F. Bantz, "PAC-MEN: Personal autonomic computing monitoring environments," in *Proc. IEEE DEXA 2004 Workshops—2nd Int. Workshop Self-Adaptive and Autonomic Computing Systems (SAACS 04)*, Zaragoza, Spain, Aug. 30–Sep. 3, 2004, pp. 737–741.
[34] R. Sterritt, B. Symth, and M. Bradley, "PACT: Personal autonomic computing tools," in *Proc. IEEE Workshop Engineering of Autonomic Systems (EASe 2005) at 12th Annu. IEEE Int. Conf. and Workshop Engineering of Computer Based Systems (ECBS 2005)*, Greenbelt, MD, Apr. 3–8, 2005, pp. 519–527.

Roy Sterritt (M'01) received the B.Sc. (Hons) degree in computing and information systems and the M.A. degree in business strategy from the University of Ulster, Newtownabbey, U.K., in 1994 and 1999, respectively.

He was a Software Engineer with IBM at North Harbour and IBM Hursley Park, U.K., developing intelligent distributed applications to support project management, risk assessment, and mobile computing. He then returned to the University of Ulster to research automated and intelligent approaches to the development and testing of fault management telecommunications systems in a series of collaborative projects with Nortel Networks. He is a faculty member at the University of Ulster and a Researcher within the Computer Science Research Institute and the Centre for Software Process Technologies. He is currently a Visiting Researcher at NASA and previously at British Telecom. He is the coauthor of over 100 technical papers on artificial intelligence, software engineering, autonomic computing and autonomic communications and has been active within the research community on program and organizing committees.

Mr. Sterritt is a Vice-Chair of the IEEE Technical Committee on Engineering of Computer Based Systems and the Founding Chair of the IEEE Task Force on Autonomous and Autonomic Systems.

David F. Bantz (M'06) received the A.B. degree in physics and the B.Sc. degree in electrical engineering in 1965, the M.Sc. degree in 1967, and the Eng. Sc.D. degree in electrical engineering and computer science in 1970, all from Columbia University, New York.

He has been an Adjunct Professor for nearly 25 years with Columbia University. In 1972, he joined IBM's T. J. Watson Research Center as a Research Staff Member and retired in 2005. Currently, he is an Adjunct Professor at the University of Southern Maine, Portland. He was an Architect of Maestro, a project in subscription computing services. He has 27 issued patents His technical interests have always been in personal computing applications and technology. His more recent work includes being architect and implementer of Prism, a rule-based Java system for the analysis of system data and for the planning of lifecycle actions.